6.831 User Interface Design & Implementation
Fall 2004

PS0: Java Swing Warmup

As a warmup for the programming assignments in this course, this problem set asks you to build a small user interface that searches a list of words. We provide a backend class that does the heavy lifting (actually loading the word list and searching it). We also specify the design of the user interface. Your job is to implement it using the Java Swing user interface toolkit.

To do this assignment, you'll need to know the following:

- how to program in Java: writing, compiling, and running Java programs
- how to use Swing widgets: windows, menubars, labels, text fields, buttons, scroll panes, and lists;
- how to use a layout manager to lay out widgets in a window automatically;
- how to use listeners to respond to user input;
- how to use a standard dialog box (JFileChooser);
- how to package and deploy a Java program in a jar file.

If any of these topics are new to you, or you want to brush up on them, here are some sources that you may find helpful:

- The Java Tutorial: a free online tutorial for the Java programming language
- Creating a GUI with JFC/Swing: the section of the Java Tutorial that concerns Java Swing
- Ivor Horton, *Beginning Java 2 -- JDK 1.4 Edition*, Wrox Press, 2002. Tutorial introduction to all parts of Java, including user interface libraries. No knowledge of other languages is assumed. Available at Quantum Books, or from amazon.com
- Bruce Eckel, *Thinking in Java*, 3rd edition, Prentice-Hall, 2002. Also available on-line at Mindview.net (but don't try printing it yourself, as it is over 1000 pages). This is written for the person who can already program, but wants to learn object-oriented thinking and the Java language. It goes into lots of detail on the tricky aspects like GUIs, multithreading, and remote method invocation.

Provided Resources

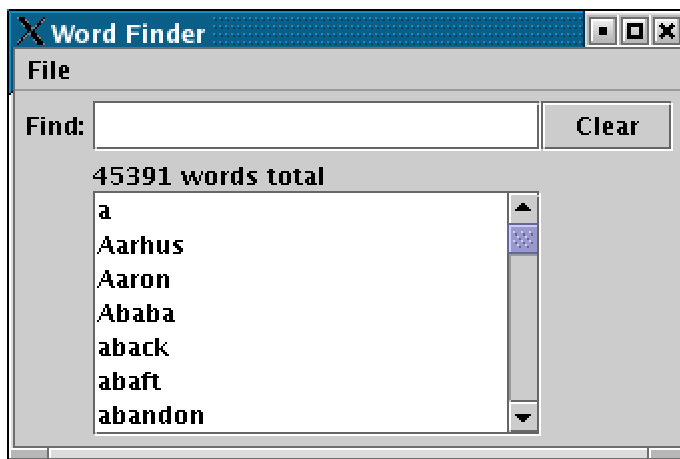
We provide you with the following:

- WordList.java: a backend class that represents a list of words and provides operations for loading it from a stream and searching it. This class includes a main method that demonstrates how to use it.
- words: a dictionary of 45,391 words taken from the standard Linux `/usr/share/dict/words`.
- WordFinder.java: a skeleton for your user interface, which contains a main method that creates an empty window.

Feel free to change these classes as you see fit.

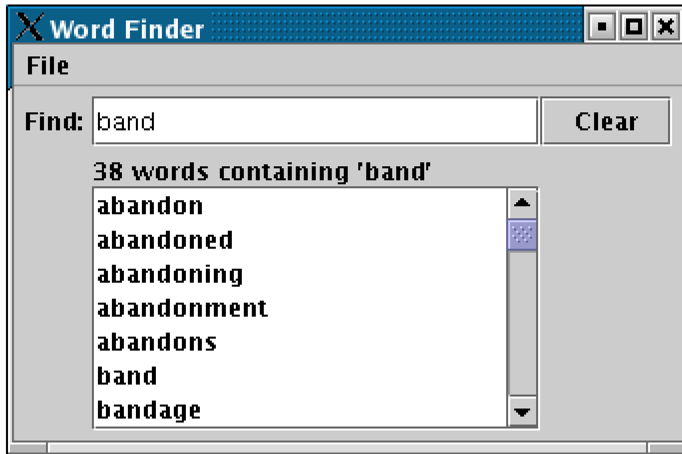
Problem

You should build an interface that looks like this:



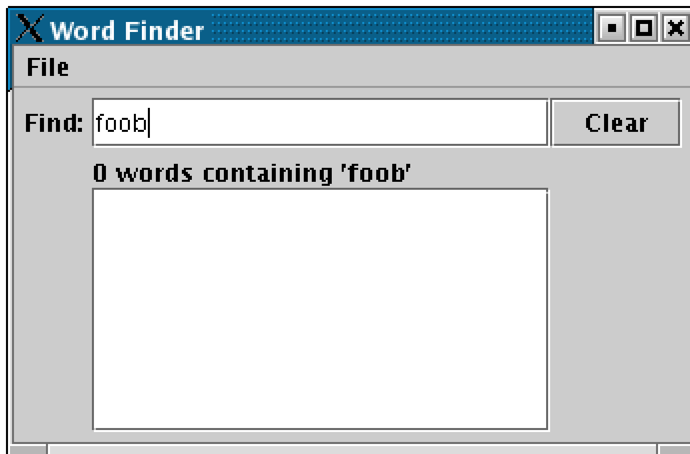
The interface should have the same layout as shown above and there should be borders between widgets. When the program is first run, the list box should display the word list we gave you. (Hint: this word list should be accessed as a *resource*, not as a file, because you will have to pack it in your jar file to hand it in. See WordList.main() for an example showing how to find the words list as a resource. The Java documentation has more about resources.)

The Find text field is the user's query. When the query is blank, the list box displays the entire word list, as shown above. Whenever the query changes, the list box immediately updates to display all words that contain the query text:

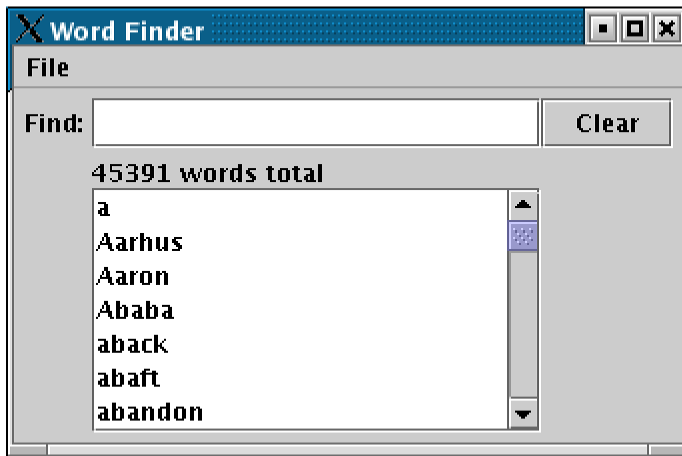


The list box should update constantly as the user types. Pressing Enter should not be necessary. (Hint: this requires you to use a listener that receives every change to the text field; see the JTextField class overview for a hint about which listener to use.)

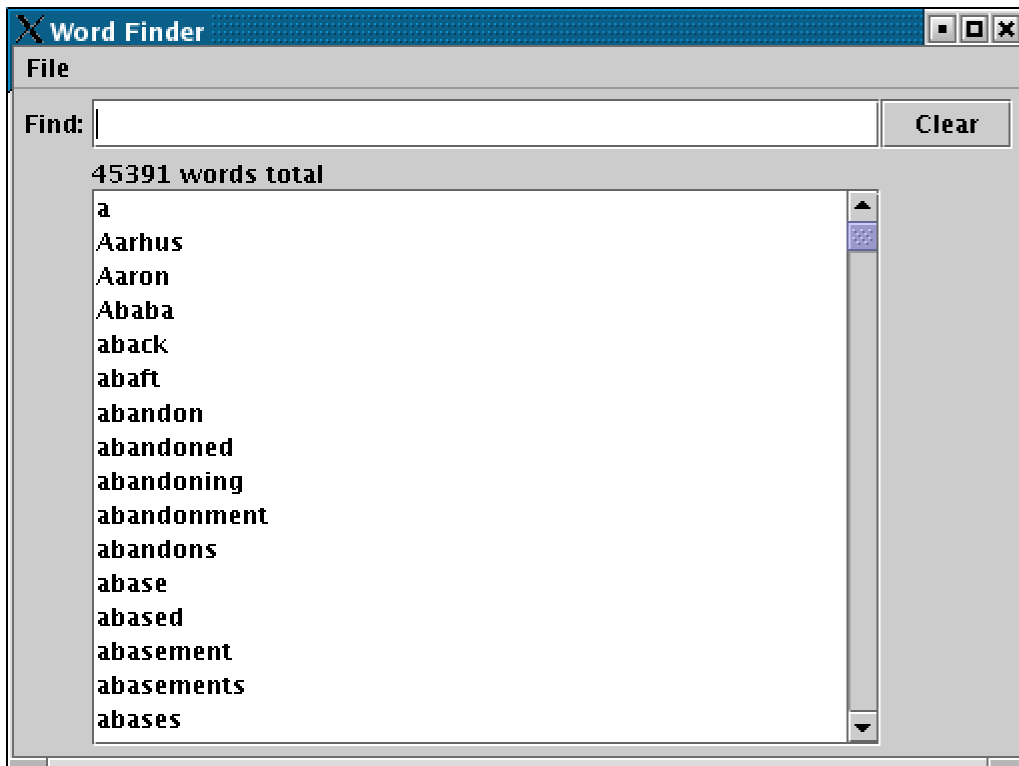
If none of the words contain the query, the list box should be empty:



The Clear button should clear the query field, restoring the list box to displaying all words again:

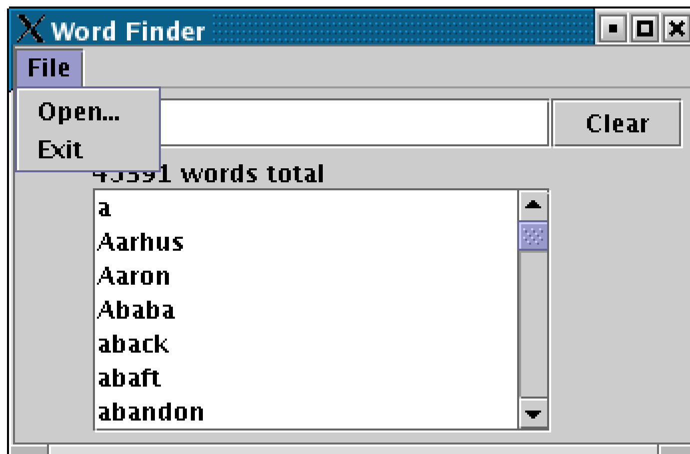


The window should be resizable, with all extra space going to the query field and the list box:



(Hint: although this layout isn't hard to create with the standard Java layout managers, you might find ClearThought's TableLayout more fun to use.)

Finally, there should be a File menu with two options:



File/Open should pop up a JFileChooser dialog box prompting the user to select a file. The words in the selected file should become the new word list. This file should be opened like a regular file from the user's filesystem --- it is not a resource like the default word list that your program loads at startup time.

File/Exit should end the program. Closing the Word Finder window should also end the program.

Questions

Answer the following questions in a README text file included in your handin.

1. Your interface provides *incremental search*, since it updates immediately as the user's query changes. An alternative approach would require the user to submit the query explicitly, by pressing Enter or clicking a button with the mouse. This approach is sometimes called *delimited search*, because an explicit action is required to delimit the end of the query. Give some reasons why incremental search might be more usable than delimited search.
2. WordList, the backend class that does the searching, is optimized for delimited search, not incremental search. Explain why. Hint: suppose the list has N words and the user enters an m-letter query. How much work would WordList have to do for a delimited search interface? How much for an incremental search interface?
3. How could you change WordList to improve its performance for incremental search?
4. What does this example suggest about how the backend implementation should be involved in the iterative design process?

Going Further

This interface is a starting point for a crossword puzzle dictionary. If you found this assignment easy and you're inclined to go further, here are some ideas for optional improvements:

- Highlight the matching substring of each word in the list box, so that the user can see at a glance how the word matches the query. (Hint: Swing controls allow HTML formatting, which makes this easy. But a good design will keep this formatting out of the WordList backend class.)
- Add controls that constrain the length of the word, so that (for example) the list box displays only 6-letter words.
- Add controls for constraining the query to the start or end of the word.
- Extend the interface with new controls and new behavior so that the user's query can be a crossword puzzle entry with some letters filled in and some letters empty.
- Display the definition of a selected word using JEditorPane and an online dictionary, such as this webster gateway.
- Eliminate the noticeable delay that occurs when the list box must display more than 10,000 words. (Hint: don't bother changing WordList as we discussed in the Questions section, because it isn't the biggest performance bottleneck. Implement your own ListModel instead. Make sure to use JList.setPrototypeCellValue() to avoid size calculations.)

What to Hand In

Package your completed assignment as a **jar file**. Your jar file must contain:

- **Class files**. Compiled class files for your program.
- **Source files**. All Java source files you wrote.
- **Resources.** All the resources your code needs to run. For example, the `words` file must be included in your jar file; so should any images or other data files your code requires. All resources of this kind should be accessed with Class.getResource(). Your code shouldn't have any absolute filenames in it, since they're not likely to work on our machines.
- **Readme.** A plain text file called `readme.txt` that answers the questions asked in this assignment and lists the people with whom you discussed the assignment.
- **Manifest.** Your jar file's manifest file must include a `Main-Class` parameter that specifies the main class, so that we can run your jar using `java -jar` *`yourfile.jar`*

The Java tutorial has a section about jar files that explains how to create a jar file and define its manifest file. If you're using Eclipse, you can create a jar file from your project using File/Export. The Export wizard offers options for including source files, readme and other resources, and specifying the Main-Class for your jar's manifest.

If your program depends on any third-party libraries, you have two choices:

- Unpack the other jars into class files, and include those class files in your jar. This is the least-error-prone approach, and is strongly recommended.
- Put a `Class-Path` parameter in your jar's manifest file specifying the jar files it depends on. For example:

```
Class-Path: TableLayout.jar xerces.jar
```

Then hand in TableLayout.jar and xerces.jar alongside your own jar file. This is error-prone because you might forget to send us a required jar file.

Before you submit your solution, put all the jar files you plan to submit in an empty directory and make sure you can run it:

```
java -jar yourfile.jar
```